h-ubu: An Industrial-Strength Service-Oriented Component Framework for JavaScript Applications

Clément Escoffier, Philippe Lalanda, Nicolas Rempulski

▶ **To cite this version:**

Clément Escoffier, Philippe Lalanda, Nicolas Rempulski. h-ubu: An Industrial-Strength Service-Oriented Component Framework for JavaScript Applications. FSE 2013 - ACM SIGSOFT Symposium on the Foundations of Software Engineering, Aug 2013, Saint Petersburg, Russia. pp.699-702. hal-00854339

HAL Id: hal-00854339
https://hal.science/hal-00854339

Submitted on 26 Aug 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

h-ubu – an industrial-strength service-oriented component framework for JavaScript applications

Clement Escoffier
Grenoble University
clement.escoffier@imag.fr

Philippe Lalanda
Grenoble University
philippe.lalanda@imag.fr

Nicolas Rempulski
Ubidreams
nrempulski@ubidreams.com

ABSTRACT
In the last years, we developed web applications requiring a large amount of JavaScript code. These web applications present adaptation requirements. In addition to platform-centric adaptation, applications have to dynamically react to external events like connectivity disruptions. Building such applications is complex and we faced sharp maintainability challenges. This paper presents h-ubu, a service-oriented component framework for JavaScript allowing building adaptive applications. h-ubu is used in industrial web applications and mobile applications. h-ubu is available in open source, as part of the OW2 Nanoko project.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Frameworks, Modules and packages, Patterns.*

General Terms
Design, Management, Languages.

Keywords
Adaptation, Service-orientation, Dynamism, JavaScript

1. INTRODUCTION
The web has undergone deep changes in the past few years. Its popularization, combined with users mobility, has drastically impacted the way web applications are built. Such applications were traditionally developed using server-side technologies such as JavaEE, PHP and ASP.Net. However, to improve usability and to benefit from the latest browsers features, new architectures where code is distributed between server and client sides have emerged. This architectural shift is not without consequences. In particular, most modern web applications contain a huge amount of JavaScript on the client side.

JavaScript is an old language[1], often despised by developers. It exhibits tricky behaviors making runtime evolutions and longer-term maintenance complex. Browsers' diversity also increases the complexity of JavaScript code. Maintaining large codebase involves dealing with browsers' incompatibilities and evolutions.

In the last four years, we developed several web applications requiring a large amount of JavaScript code. These applications were characterized by stringent adaptation requirements. Indeed, adaptation capabilities were necessary in order to deal with heterogeneous running platforms (browsers features, screen size) and to react smoothly to Internet connection losses. Such adaptations are complex to implement and involve a large amount of tricky code. To keep our technical debt under control, we decided to develop a service oriented component framework for JavaScript inspired from iPOJO[2] and CDI[3].

This decision was in line with the most recent trends in web application development. Indeed, a number of JavaScript component frameworks have emerged to make web applications easier to develop and maintain. Some of them allow developers to better structure their code into modules such as require.js (requires.org) and CommonJs (commonjs.org). However, modules are strongly coupled, which prevent easy architectural adaptations. A second approach towards componentization relies on MVC[4] and MVVM[5] frameworks. Numerous frameworks, like angular.js (angularjs.org) and backbone.js (backbonejs.org), allow to structure applications according to these architectural styles. Applications developed with those frameworks present reduced maintenance costs, but also exhibit incompatibilities with other frameworks. Introducing complex logic and adaptability is also difficult in these frameworks, focused on UI implementation.

In this paper, we present h-ubu, a service-oriented component framework for JavaScript applications. The purpose of this framework is to bring modularity to JavaScript code but also runtime adaptability. This framework was successfully used in several industrial web and mobile applications. One of these use-cases is presented in the paper. A study of the impact of h-ubu on the development is presented as well.

2. SERVICE-ORIENTED COMPONENT MODELS TO SUPPORT ADAPTATION
Service-oriented component models come from the infusion of service-orientation[6] inside component models. Unlike traditional component models where components are linked before execution, service-oriented component models promote runtime resolution of service dependencies[7]. Components offer and require services described as service specifications. This class of component models was popularized by Service Component Architecture[8]. To resolve service dependencies, components look up inside a service registry and select a service provider. One interest of such approach is the loose-coupling resulting from the reduced amount of data shared between providers and consumer.

Service-orientation fits well to dynamic environments as services can arrive and leave anytime. Such dynamism is a key concern in web applications as remote services can become unreachable when the Internet connection disrupts. In addition, components can select their provider according to the execution environment, such as the platform. Services can also reify the browser capabilities. Providers have the ability to withdraw an exposed service from the service registry if the execution context does not meet their needs anymore.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia.
Copyright 2013 ACM 978-1-4503-2237-9/13/08... $15.00.

In addition to the adequacy between the service paradigm and the dynamic variability of web execution environment, service-oriented component models offer several advantages during development. As the application is decomposed into well-defined components, the development can be easily shared among developers and each component can be tested separately.

3. H-UBU

h-ubu is a component framework applying the service-oriented approach to JavaScript. Its purpose is to bring modularity to applications and to ease their runtime adaptation. It also leads to a reduced amount of code since important features are addressed by the framework itself. h-ubu is based on the notion of components with provided and required services, and on a hub, a specific component in charge with runtime components bindings.

3.1 Service-oriented components

h-ubu's applications are built from service-oriented components. These components are developed in JavaScript as JavaScript object (see code sample 1). To be managed by the h-ubu framework, they must possess three lifecycle callbacks: *configure, start* and *stop.* The *configure* method contains the code describing required and provided services. *Start* and *stop* methods are called by the framework when a component is started, resp. stopped.

```
var minimalComponent = {
    configure : function(hub, conf) {
        this.hub = hub;
        this.name = conf.name;
     },
    start : function() { },
    stop : function() { }
}
```

Code sample 1 - A h-ubu component and its lifecycle callbacks

h-ubu components interact through services, described with *service contracts*. Contracts are also JavaScript objects (see code sample 2). They define a set of methods (with empty bodies) and properties.

```
var myContract = {
    doSomething: function() {}
};
```

Code sample 2 - Example of service contract

To publish and require services without code overhead involved in dynamic service-orientation, h-ubu proposes a declarative approach. Indeed, tracking services and properly handling interactions with the service registry can be cumbersome. As stated above, components describe their provided and required services within the *configure* method.

As illustrated in the following code, declaring a provided service is done with the *provideService* method. The developer provides the service contract and optionally a set of service properties.

```
var component = {
    configure: function(hub, configuration) {
        hub.provideService({
          contract : myContract
          properties: {property: "value"}
        });
    },
    doSomething: function() { … } // Contract implementation
    //…
}
```

Code sample 3 - A component declaring a service

Components providing services must conform to the contracts, *i.e* implement all methods and publish the contract's properties as service properties. The h-ubu framework verifies this conformity.

Despite this declarative way, components may need to act on their provided services directly. The *provideService* function returns an object allowing components to decide whether the service must be published or not. This control over service publication is very handy when a component needs to check whether the execution environment meets its expectations to serve its service.

To declare a service dependency, a component calls the *requireService* method. Service dependencies are characterized by several attributes, including:

- The target service contract,
- The optionality and cardinality of the dependency. By default, dependencies are mandatory and scalar,
- An optional filter to select a service provider,
- The type of injection.

When declaring a service dependency, a component chooses between three injection mechanisms: injection inside a component field, injection using callbacks (called when a service arrives and leaves), or no injection. As illustrated by code sample 4, field injection has the less impact on the code: the service can be directly used from the injected field. Callbacks allow tracking service events. When a service provider is arriving or is leaving, the component is notified and can react. Finally, the last mechanism let the component ask for an immediate lookup using the *locateService* method. It does not track the availability of the service, so the component must be prepared to receive a *null* object. As stated in [9], those mechanisms cover the high majority of the use case.

```
var component = {
    configure: function(hub, configuration) {
        hub.requireService({
        contract : myContract, // service contract
        field : "mysvc" // field injection
      });
    },
    //…
    work : function() {
         // using services does not require additional code
         this.mysvc.doSomething();
    }
}
```

Code sample 4 - A component consuming a service

Components using services have only access to the methods from the service contracts. Thanks to a proxy generation mechanism, h-ubu is able to enforce this aspect. Thus, h-ubu applications do not suffer from the visibility issue of JavaScript.

3.2 The hub

The hub can be seen as a dock for application components. At runtime, the hub manages plugged components. This includes the component's lifecycle and the weaving of bindings between components. Communication follows service-orientation to support runtime adaptation. At runtime, the hub is in charge of publishing, tracking and injecting services.

As shown in code sample 5, components are registered on hubs when the web page is loaded. Components receive the hub object on which they are plugged and an optional configuration. This configuration contain, for instance, a selector on the page element they need to handle. This externalization improves the reuse of components between pages and applications. It also allows the implementation of applications following the MVC or MVVM patterns.

```
hub
   .registerComponent(component1)
   .registerComponent(component2, {
        list: "#list"
   })
   .start();
```

Code sample 5 - Component registration

The hub manages a service registry as well as a message-oriented middleware to support communication between components. To enable hierarchical composition, hubs are also components and so can be registered on a parent hub. Those hubs can import and export services from and to the parent hub (see figure 1). Components from the sub-hub have only access to services from components plugged on the same hub and imported services. In addition, services from the sub-hub are not available from the parent hub, except if they are exported. This architecture style is close to C2ADL[10], and Enterprise Service Bus[11], but focused on JavaScript.

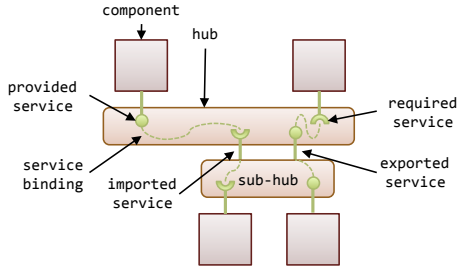

Figure 1. h-ubu application architecture

3.3 Runtime Management and Adaptability

h-ubu components are executed within a container dealing with service tracking, injection, and publication (figure 2). This container is configured in the *configure* method of the component.

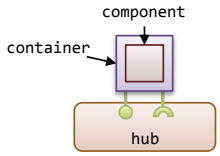

Figure 2. h-ubu's container

When the application is started, the containers initialize the service tracking. When all the service dependencies of a component are satisfied, the container registers the provided services.

If, on a service departure, a mandatory service dependency becomes unsatisfied, the container unregisters the service published by the component. They will be registered again when the service dependency will become fulfilled.

Such component-centric dynamic management allows the applications to exhibit dynamic adaptability thanks to a cascading effect. As an example, when the Internet connection is lost, the component tracking the connection withdraws its services. All components requiring the connection also unregister their services from the service registry. Facing this change, the component implementing the UI can decide to disable features requiring Internet connection or notify the user. As soon as the Internet connection is restored, the service is published again.

Adaptation can also be platform-based. Components can decide to publish or not their service according to their execution environment. Thus, a component may decide whether to publish its service according to the availability of browser features (like (geo-localization, SVG support or local storage). So a h-ubu application can have different configurations on different platforms. This ability is critical to support web applications running on a large set of devices (laptop, mobile phones and tablets) and browsers (Firefox, Chrome, Internet Explorer).

4. INDUSTRIAL APPLICATION

4.1 Context

This section describes an application developed by Ubidreams (http://ubidreams.com/) for one of its customers, using h-ubu for variability and adaptability.

Precisely, Ubidreams developed a product catalog (see figure 3) accessible from regular browsers and from iOS devices (as a native application). The feature set differs in the web site and in the iOS application and also depends on the user's roles. The web site offers a management system where authenticated users can add, update and remove products. This part of the application is a business-driven CMS developed for the project. The iOS application focuses on products browsing and viewing.

Figure 3. The gourmandise application

The iOS application is used by salesmen for face-to-face meetings with customers. They must have up to date products but at the same time can face Internet connection disruption. To deal with that, the application locally stores the set of products and syncs when an Internet connection is available. As the catalog is presented to customers, it displays a modern, fluid and attractive user interface. The application is used in production.

4.2 Configuration variability

As presented above, the set of features proposed by the application heavily depends on the platforms (browser or iOS device) and the users. We identified two types of variability:

1. The application has different purposes for the CMS and for the iOS devices. But lots of code is common.
2. The feature set changes depending on specific user roles (administrator, editor, viewer)

The web site and the iOS application are composed by different sets of components. However, thanks to h-ubu, a large amount of components are reused. Precisely, the iOS applications are made of 61 components, while the administration part contains 56 components. 29 components are shared between the two parts. In addition, the web site and iOS application share a common set of services. However, implementations differ as the execution environments are different. For instance, both parts are using a local storage service. The web site implementation relies on the browser local storage, while the iOS application wraps the application file system. The implemented service contract is identic, enabling the reuse of consumer's components.

According to the user role the set of available features also change. Such conditional configuration is implemented directly by the components themselves. Components decide whether to publish their service. Thus, the set of published service is different

and changeable. When a user logs in, components react and register their services. Thanks to these dynamic changes, the user interface is updated seamlessly, without page reloading.

4.3 Dynamic adaptation

The iOS application is used in mobility. Remote services can become unreachable due to connectivity disruption. Implementing dynamism by making these services available or unavailable makes building adaptable application easier.

The iOS application is based on a set of dynamic services. The application adapts itself according to the available set of services. For instance, the authentication service is implemented by a component that tracks the availability of the backend authentication service. When this service is not reachable, the component unregisters the service. The application reacts by hiding all actions requiring authentication. Products are also synchronized by the application and stored locally. The sync process is triggered periodically in background when the Internet connection is established. If syncing fails, it is re-triggered as soon as the connectivity is reestablished.

4.4 Metrics and Feedbacks

The application presented in this section is composed by 31744 line of code (JavaScript and HTML). This number does not include the backend application nor the libraries and components integrated in the application. The code is structured into 88 components assembled differently according to the system.

The application runs without performance penalty. In production, the proxy injection is disabled, reducing the overhead to be unsubstantial. The application load and launch time are subjected to a minimal overhead (less than 5%). This is because of the code division into more files and the service stabilization on startup.

The application was developed in 12 weeks by a small team of 4 members. The developers didn't have any experience in componentization and never used h-ubu. They, as most developers, feared JavaScript development. The simple model proposed by h-ubu made them productive after a few days. In addition, the isolation of all components has made the decomposition of the work natural. Each component was tested separately, generating confidence regarding the final application.

The modularization of the application could have been finer. The lack of experience of service-orientation and componentization has impacted the architectural choices. The team admitted that smaller components and more services would have improved code quality and increased their velocity.

However, tooling, especially build and test tools can be improved. After having analyzed the usage of h-ubu in this project, we start foreseeing guidelines and common practices. Understanding these new patterns would improve the modularization of JavaScript applications and their dynamic adaptation.

5. CONCLUSION

Mobility and HTML5 are opening a new era for web applications. However developing such complex, attractive and reactive applications is not without a high cost. JavaScript is now heavily used in all these applications. JavaScript code has the reputation to be unmaintainable and exhibits tricky behaviors. In addition these new applications often require runtime adaptations, to react to user actions and environmental changes.

In the last years, we developed several web applications relying on a large amount of JavaScript code. To keep a decent technical debt, we developed a service-oriented component framework, named h-ubu. Its purpose is to bring modularity to applications and to ease their runtime adaptation. It also leads to a reduced amount of code since important features are addressed by the framework itself. h-ubu is based on the notion of components with provided and required services, and on a hub, a specific component in charge with runtime components bindings.

As presented in this paper, h-ubu is successfully used in several industrial applications. h-ubu allows a higher code reuse, improves the code quality, and supports fine-grained modularity.

h-ubu does not aim to manage dynamic provisioning of components. These aspects are managed by *loaders,* such as Google Loader or JQuery. They are fully functional with h-ubu.

h-ubu is part of the OW2 Nanoko project. The code is licensed under the Apache License Software 2.0, and is available on http://nanoko-project.github.io/h-ubu/release/.

6. REFERENCES

[1] ECMA International, "ECMAScript - ECMA-262, 1st edition," ECMA International, 1997.

[2] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," presented at the IEEE International Conference on Services Computing, 2007, pp. 474–481.

[3] G. King, "Contexts and Dependency Injection in Java EE 6," *Contexts and Dependency Injection in Java EE 6*, 2009.

[4] R. Tanikella, G. Matos, G. Tai, and B. Wehrwein, "Relating requirements to a user interface architecture for a rich enterprise web application," presented at the 2nd international conference on Trends in enterprise application architecture, 2006.

[5] J. Freeman, J. Järvi, and G. Foust, "HotDrink: a library for web user interfaces," presented at the 11th International Conference on Generative Programming and Component Engineering, 2012.

[6] M. P. Papazoglou, "Service-Oriented Computing : Concepts , Characteristics and Directions," *Information Systems Journal*, vol. 3, no. 10, pp. 3–12, 2003.

[7] H. Cervantes and R. S. Hall, "Autonomous adaptation to dynamic availability using a service-oriented component model," presented at the 26th International Conference on Software Engineering, 2004, pp. 614–623.

[8] OASIS, "Service Component Architecture," OASIS. http://oasis-opencsa.org/sca

[9] C. Escoffier, P. Bourret, and P. Lalanda, "Describing dynamism in service dependencies," presented at the 10th International Conference on Services Computing, 2013.

[10] N. Medvidovic, P. Oreizy, and R. N. Taylor, "Reuse of Off-the-Shelf Components in C2-Style Architectures," presented at the International Conference on Software Engineering, 1997.

[11] M. Luo and L.-J. Zhang, "Practical SOA: Service Modeling, Enterprise Service Bus and Governance," *IEEE International Conference on Web Services*, p. 11, 2008.